Open-source Defect Injection Benchmark Testbed for the Evaluation of Testing

Miroslav Bures
Dept. of Computer Science
FEE, CTU in Prague
Prague, Czech Republic
miroslav.bures@fel.cvut.cz

Pavel Herout
Dept. of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic
herout@kiv.zcu.cz

Bestoun S. Ahmed
Dept. Mathematics & Comp. Science
Karlstad University
Karlstad, Sweden
bestoun@kau.se

***Abstract*—A natural method to evaluate the effectiveness of a testing technique is to measure the defect detection rate when applying the created test cases. Here, real or artificial software defects can be injected into the source code of software. For a more extensive evaluation, injection of artificial defects is usually needed and can be performed via mutation testing using code mutation operators. However, to simulate complex defects arising from a misunderstanding of design specifications, mutation testing might reach its limit in some cases. In this paper, we present an open-source benchmark testbed application that employs a complement method of artificial defect injection. The application is compiled after artificial defects are injected into its source code from predefined building blocks. The majority of the functions and user interface elements are covered by creating front-end-based automated test cases that can be used in experiments.**

***Index Terms*—Software Testing, Fault injection, mutation testing, benchmarking**

I. Introduction

To evaluate the effectiveness of a testing technique for software systems, various approaches can be employed. A natural and well-known approach to assess the effectiveness of a test suite generated by a testing technique is to measure the defect detection rate when applying a generated test suite to a System Under Test (SUT). As such, an experimental SUT that represents a real-world system containing real defects from the past software development process can be useful here. Alternatively, the mutation testing technique can be applied by introducing artificial defects into the code of an experimental SUT using defined mutation operators [1], [2]. Additionally, a defect injection technique, which can be considered to be a more general variant, can be employed. In defect injection, defects are introduced into an experimental SUT and various technical possibilities can be used.

Measuring the defect detection rate can be used to determine the effectiveness of the Combinatorial or Constrained Interaction Testing [3], [4] or Path-based Testing [5] techniques. As a typical example, one can examine the strength of test cases generated by the Combinatorial Interaction Testing (CIT) algorithm using a mutation testing technique. As an experimental SUT, an open-source software system can be selected. Then, various mutants are created from the source code by a set of mutation operators. Subsequently, the generated test cases are assessed in the experimental SUT and it is determined whether the test case can detect a defect introduced into SUT by a mutation operator. These types of experiments are typically run in multiple series with various sets of mutants and test cases to obtain convincing evidence regarding the effectiveness of the generated test cases [2].

The approach can be generalized as illustrated in Figure 1.

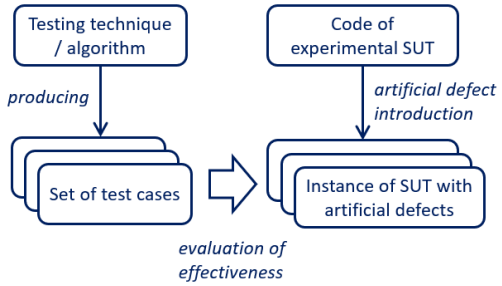

Fig. 1. Defect introduction process to an experimental SUT.

In this paper, we present a new open-source benchmark testbed to support defect injection testing. The testbed is available to the community and can be used to evaluate various testing techniques. In this approach, we do not insist on defined mutation operators. The goal of the testbed is to provide a complement to the classical mutation testing approach for evaluating the effectiveness of test cases.

In contrast to the established classical code mutation operators, various complex software defects can be introduced into the code, especially defects caused by a misunderstanding of the SUT design specification or requirements during the development process. The practical use case of the presented testbed is to provide researchers with a complementary option to the mutation testing technique to be able to simulate a broader spectrum of possible software defects during experiments. The testbed is, hence, a complement to mutation testing rather a replacement of mutation testing via a defect injection approach. As we show later in Section II, both approaches have certain advantages and disadvantages. Hence, both approaches can be combined to provide the best objective measurement of the effectiveness of a testing technique.

978-1-7281-5778-8/20/$31.00 ©2020 IEEE
DOI 10.1109/ICST46399.2020.00059

The rest of this paper is organized as follows. Section II discusses the background in more depth and analyzes the state of the art. Section III describes the presented testbed from various viewpoints, including the system scope, implementation details, available automated tests, mechanism of insertion of artificial defects and process of evaluating the effectiveness of the examined testing techniques. Section IV discusses the presented concept and also analyzes its possible limits. The last section concludes the paper.

II. Background and State of The Art

As mentioned previously, a common practice to evaluate a set of test cases generated by an algorithm is to assess the defect detection rate of the test cases in an experimental SUT that contains defects. In this general approach, several aspects have to be maintained to give the technical possibility of conducting a well-defined and objective experiment. The following bullet-points address three common aspects in this direction:

1) The defects presented in the experimental SUT simulate real defects in software projects.
2) It is possible to create a set of various instances of an experimental SUT with different sets of injected defects to examine the testing technique for a reliable sample of situations.
3) The experimental SUT has to support effective automated evaluation of the examined test cases. Hence, the experiments can be repeated with different sets of defects in an effective manner to assess more extensive sets of situations.

Table I presents an analysis of these aspects for three possibilities of artificial defect introduction within an experimental SUT. These possibilities are as follows: (1) using real project defects, (2) mutation testing, and (3) defect injection. Defect injection, here, is a generalized method in which we do not employ standard source code mutation operators. In fact, it is difficult to reach a clear understanding of an objective approach from all three discussed options when considering all the advantages and disadvantages presented in Table I. Instead, it is worthwhile to consider a combination of the presented approaches to increase the reliability of the experiments.

Among the discussed approaches, mutation testing can be considered to be the most established approach, originating in the late 70s [6]. On the technical level, this approach depends on a particular programming language. However, code mutations have been performed for major programming languages. As an example, the Mujava system [7] is used for the Java programming language and MuCPP [8] is used for C++. Here, for a particular program code mutation, a set of established operators is defined [1], [2]. While these operators are useful, there are concerns in the literature about the relation of the code mutants to real software defects and types of software defects that are difficult to express using various mutants [9], [10]. To overcome this problem, various approaches have been considered in the literature – for instance, the construction of more complex mutants [9]. However, the mutation testing approach might still meet its limit when trying to insert certain types of complex defects that may be caused by a misunderstanding of the design specification [10]. Generally, the similarity of mutants to real defects varies in empirical experiments [10], [11].

The defect injection method can be seen as a more general method than mutation testing to insert artificial defects into experimental software. In this process, various techniques at any software level can be used to insert defects, e.g., [12], [13]. As an alternative to mutation testing and artificial defect injection, a number of experiments have also been conducted using real defects from past software projects, e.g., [14]. Here, comparing those different approaches is challenging because the objectivity of the experiment in which we evaluate the effectiveness of the testing techniques strongly depends on the testing technique, the characteristics of the software used as a benchmark, and how realistic the inserted defects are compared to real defects. Moreover, the characteristics of the software defects might also change with changes in the development styles, the usage of integrated development environments and the best practices of programming. To this end, in this paper, we suggest applying a benchmark testbed as a complement to the mutation testing approach.

III. Testbed Description

To create a benchmark testbed for the evaluation of the effectiveness of the test technique, we designed and implemented the University Information System Testbed (TbUIS)[1]. The testbed is an open-source testbed that can be used to evaluate any test technique. The TbUIS system is a three-layered web application that uses a relational database as a persistent data storage and object-relational mapping (ORM) layer.

The system supports the artificial defect injection approach, as discussed in Table I (column *Defect injection*). A special module allows the creation of defect clones of the system by introducing defects from a catalog of predefined defect types as well as creating customized artificial defects to be inserted into the SUT. As a demonstration and for a quick start for experiments, a set of 28 already assembled defect clones are available for testbed users.

In this section, we describe the following aspects of the TbUIS: (A) the scope of the system and its use cases, (B) the implementation and technical details, (C) the available automated tests to be employed in the experiments, (D) the mechanism for introducing artificial defects in the system and (E) the test case effectiveness evaluation process, including the logging mechanism used in the evaluations.

A. Scope and Use Cases of the TbUIS

The TbUIS is a fictional university study information system that supports a study agenda related to students' enrolment in courses, management of exams and related processes. The standard actors of the system are students and lecturers.

[1] https://projects.kiv.zcu.cz/tbuis/

TABLE I
BRIEF COMPARISON OF ARTIFICIAL DEFECT INTRODUCTION TYPES TO AN EXPERIMENTAL SUT

Discussed aspects	**Defect introduction method**		
	Historic defects	***Mutation testing***	***Defect injection***
The objectivity of the defects[a]	The defects correspond to a real software project; however, the used sample of defects can be limited, which can restrict the objectivity of the experiment to only one particular experience-based case.	Various combinations of mutation operators can be selected. This approach allows the flexible mixing of various defects made by the programmer. More complex defects caused by a misunderstanding of the specification can be simulated by a set of mutation operators.	More complex simulated defects are not limited to a defined set of mutation operators. Additionally, it might be difficult to prove that an artificially elaborated defect is likely to occur in the real software development process.
Ease to create instances[b]	In some cases, creating multiple instances might be challenging, as there are a limited number of defects from the past software development process.	Technically, creating new mutants is straightforward, and the number of various created SUT instances is practically unlimited.	If a set of artificially elaborated defects is limited, then the possible number of instances of experimental SUTs that can be configured is limited.
Test automation coverage[c]	Test automation options are not influenced by a particular defect introduction method; automated testability is rather influenced by the structure and coding standards employed in an experimental SUT		

[a]How the introduced defects are realistic in comparison to real current software development process
[b]How easy is it to create an extensive set of various configurations of an experimental SUT with different inserted defects
[c]How easy is it to cover an experimental SUT by automated tests that help to evaluate whether the examined test scenarios detect an inserted defect

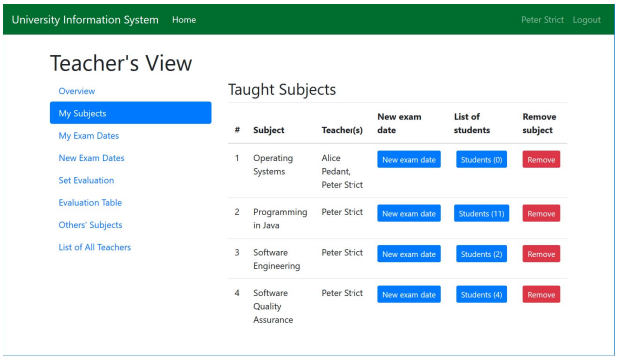

Fig. 2. Example of TbUIS user interface—lecturer's view.

The whole system can be summarized into 21 general, high-level use cases. Five use cases are related to a user who is not logged in. Another two use cases are common for lecturers and students and cover the login mechanism and user settings. The students' part is then defined by five use cases and the lecturers' part by nine separate use cases.

The graphical layout of the user interface (UI) of the system is kept relatively simple and compact, considering the goal of the system, which is to evaluate testing techniques as well as the need to cover the application by reliable front-end (FE) based automated tests to support this process (introduced later in Section III-C). An example of the TbUIS user interface is presented in Figure 2.

Regarding the process flow, the possible states and functions of the system are documented in the UML Activity Diagram schema in the Oxygen[2] [5] application and are available in the Oxygen project format as well as the SVG graphical format. This model of the current version of the TbUIS is composed of 119 different states and 164 transitions among them.

[2]http://still.felk.cvut.cz/oxygen/

TABLE II
SIZE OF TBUIS SOURCE CODE

	Number of files	Size of files [KB]	LOC
Java	87	340	8550
JSP	18	94	1550
total	105	434	10100

B. Implementation and Technical Details

Technically, the TbUIS is a layered web-based application implemented in J2EE with Java Server Pages (JSP) and Spring. As the ORM layer, Hibernate is used. For the implementation of the UI, Bootstrap is used.

In the user interface of the TbUIS, all the common basic types of control for web elements (e.g., menus, buttons, check boxes, selections, modal windows, etc.) are used. Each element (including rows in tables) has its own unique ID attribute to ease the creation of FE-based functional automated tests.

The extent of the TbUIS source code is documented in Table II. Here, the number of source files, size of source code files in kilobytes and number of lines of code (LOC) are presented separately for back-end code in Java as well as for UI code in JSP. Unit tests as well as functional automated tests are not included in these statistics.

Any important activity in the TbUIS testbed is reported in detailed application logs implemented by Log4J2. Because of the logging framework configuration options, the user can customize the level of detail and the output stream of the log. The application logs are also extended by the activation information of the inserted artificial defects (further discussed in Section III-D) and can be paired with the logs of available functional automated tests (further discussed in Section III-C).

C. Automated tests

TbUIS is strongly covered by various types of automated tests that have the following two goals:

TABLE III
EXTENT OF SOURCE CODE OF AUTOMATED TESTS

	Number of files	Size of files [KB]	LOC
Unit tests for TbUIS code	33	277	6945
Shared libraries for FE-based functional tests	101	452	14707
Unit tests for shared libraries for FE-based functional tests	30	97	2649
FE-based functional tests for TbUIS	81	248	7530
total	245	1024	31831

1) To ensure that the system (before the introduction of controlled artificial defects used to evaluate the effectiveness of testing techniques) is largely free of other defects and
2) To support the process of evaluating the effectiveness of the testing techniques by executing the defined test cases that are to be examined in the system via automated tests,

Two types of tests are available as extra modules for the TbUIS testbed:

1) **Unit tests** implemented in the JUnit framework, which test individual methods of the system and the basic sequences of methods calls on the technical level.
2) **FE-based functional tests**, which simulate users' tests accessing the system UI. These tests are written in Java with the Selenium Web Driver API, currently version 3.141.59. The tests are structured using the PageObject pattern, which significantly decreases their maintenance and allows future extensions of the test set, as independently verified [15].

Regarding the coverage level, in the current version of the TbUIS, the line coverage of the available unit tests is greater than 85%.

The FE-based functional tests cover all of the processes, as documented in the process flow schema created in the Oxygen application (introduced above in Section III-A).

To determine the expected test results of the FE-based functional tests, the Oracle module is implemented and is thoroughly tested using a special set of unit tests.

Table III provides insight into the extent of the implemented automated tests. The number of source code files, their size in kilobytes and the number of lines of code (LOC) are presented. The FE-based functional tests for TbUIS employ several modules of reusable objects and support code (in Table III denoted as *Shared libraries for FE-based functional tests*). These modules are also covered by their own set of unit tests.

Compared to size of the source code of the TbUIS (see Table II), the extent of the automated tests measured in terms of LOC is approximately three times higher.

The FE-based functional automated tests are divided into several types, covering various technical and user aspects of the TbUIS:

- **Atomic tests** that are verifying if elements of application UI are correctly rendered and filled with correct data
- **Process tests** that are exercising individual processes in the TbUIS (e.g. enrolling a course or assigning a grade to the student)
- **Negative tests** that are testing boundary conditions and correct handling of wrong input data

TABLE IV
TYPES OF FE-BASED FUNCTIONAL AUTOMATED TESTS

	Number of tests	Number of asserts	Elapsed time [sec]
Atomic tests	890	2702	780
Process tests	64	2351	1477
Negative tests	29	52	50
total	983	5105	2307

Atomic types of tests are also orchestrated as parts of the process tests. The test scripts are organized into building blocks that allow the automated composition of an automated end-to-end test via a defined path-based test scenario (the details are presented in Section III-E).

The numbers of tests in the individual categories with their numbers of asserts and average runtime are presented in Table IV. The runtimes were measured using the following configuration: Intel i5 1.6 GHz, 16 GB RAM, MS Windows 10pro operating system, Apache Tomcat 9.0 application server and MySQL database. The database and web and application servers were installed on the same workstation, and the automated tests were run on the same computer.

The automated atomic tests cover 100% of all active and passive elements composing the user interface of the TbUIS. As active elements, we consider user control elements (e.g., text boxes, drop-down menus, links, etc.) and fields that display data loaded from the database or are taken from the runtime memory of the application. Each of the active elements is tested at least by one atomic test.

FE-based automated functional tests can be easily run from a special application, TestRunner, which provides its own user interface in which particular tests to run can be selected. The TestRunner application can be downloaded from the project web page.

The extent of the building blocks of the FE-based automated functional tests introduced in this section allows the effective composition of automated tests for the path-based test scenarios to be evaluated in the testbed. The relevant part of these blocks can also be used to evaluate the combinatorial or constrained interaction testing test sets.

D. Introduction of Artificial Defects

Artificial defects are introduced into the TbUIS by the error seeder module, which conducts the following process:

1) The error seeder takes the baseline TbUIS code, which is considered free of defects (which is verified by the thorough automated tests introduced in Section III-C).

2) Based on the artificial defect specification, the error seeder assembles the source code of a defect clone of the TbUIS.
3) Then, the defect clone of the TbUIS system is built and deployed to a testing environment.

Artificial defect specification defines a set of artificial defects that are to be introduced into the TbUIS code. Predefined catalogue defect types are available as well as the possibility to define custom artificial defects. The catalogue of defect types is available on the project web page.

Each artificial defect inserted into the TbUIS code is accompanied by a logging mechanism that records information if and when the defect has been activated by a test. The main purpose of this information is to support the evaluation of the effectiveness of the testing techniques. The defect activation logs can be paired with the logs of available automated tests to give reliable sources of information, which artificial defect were detected by which test cases.

In the current version of the TbUIS testbeds, a set of 27 artificial defects of various types from the above catalogue is available for initial experiments and are accompanied by detailed information making their application easy[3].

As mentioned above, for further experiments and to evaluate the effectiveness of the testing techniques, more various defect clones can be created and compiled from available artificial defects, and also, based on the well-documented examples in the source code, the user can implement their own artificial defects.

E. Test Case Effectiveness Evaluation Process

As introduced above, in principle, the effectiveness of various testing techniques can be evaluated in the TbUIS testbed. In the following section, we focus on two major representatives, path-based techniques and combinatorial/constrained interaction testing techniques.

The parts of the TbUIS testbed related to the evaluation of path-based testing techniques are summarized in Figure 3. The inputs and outputs of the process are depicted by yellow boxes.

The input to the process is a *path-based test set*, whose effectiveness is going to be evaluated. The test cases in this test set have to correspond to an available *TbUIS process model* (unless we intentionally created invalid paths-based test cases in the experiment). Using predefined building blocks from the *FE-based functional automated test scripts* (introduced in Section III-C), the *process test builder* chains these building blocks as instructed by the input path-based test cases to produce *assembled FE automated tests*, which represent individual path-based test cases. For each of the path-based test cases at the input, a corresponding automated FE test is created.

The second input of the process is *artificial defect specification*, which can be created via predefined *catalogue defect types* or *own defects* defined in the SUT. To create an *defect clone* of the TbUIS with the specified defects, *Error seeder* takes the specification of the defects and inserts them into the code of the Baseline UIS specification. Then, the defect clone is built as a running system instance.

[3]https://projects.kiv.zcu.cz/tbuis/web/page/download

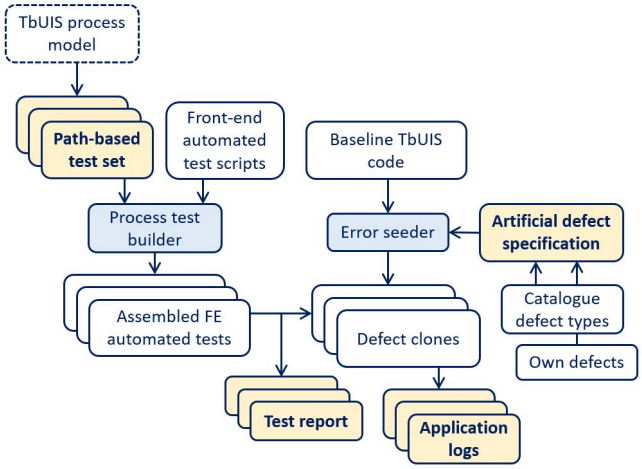

Fig. 3. TbUIS parts for evaluation of path-based testing techniques.

At this stage, experimental evaluation of the path-based test set can be performed (an example is provided on the project web pages). Automated FE tests corresponding to the input path-based test cases are run in the defect clone, and the results are reported to the *test report*, which can be evaluated. The information from the test report can also be paired with detailed *application logs* to obtain more context information about the activated defects.

The schema for evaluating combinatorial or constrained testing test cases slightly differs, but the general principle remains the same.

In this type of evaluation, we do not compile FE automated tests to correspond to path-based test cases; instead, we can use

1) available automated FE-based functional tests covering all active elements and processes in the TbUIS,
2) available unit tests available together with the TbUIS code, or
3) combinations of both types of tests (the automated tests available to the TbUIS were introduced in Section III-C).

Input data combinations to be exercised in the testbed can be entered into the available automated tests via the standardized DataProvider interface of the JUnit framework.

IV. Discussion and Possible Limits

Like other alternative artificial defect introduction approaches discussed in this paper, namely, using real defects from a previous software project and code mutation, the approach taken in the proposed testbed has certain advantages and disadvantages. We summarize these advantages and disadvantages in this section.

Regarding the possible complexity of the artificial defects introduced into an experimental SUT, the proposed approach

does not limit an artificial defect to a set of mutation operators or a conditionally switched block of code. Instead, the defect clone can be built with the changes made in several different places in the source code, which allows high flexibility in simulating complex defects.

Concern whether the introduced defects represent typical defects that are being made during real software projects can be raised. This responsibility in experiments is up to the researchers and testing practitioners. Typical defects might vary between various software architectures, development styles, programming languages, business domains, and even decades when the empirical observations are made. Hence, the testbed provides a general possibility to create different types of defects and defect clones, and the decision is up to the testbed user.

In the proposed concept, the artificial defects are selected from a pre-defined set, which might limit the generalization of experiment results. This potential limit can be solved by the addition of more artificial defects as well as the correct interpretation of the results of the experiments.

Also, certain defects might be easier to detect than other defects, which may impact the results of the experiments [16]. However, this concern can be raised generally for any defect injection technique and shall be mitigated by correct interpretation of the results of the experiments.

Another concern is that the system is artificially created; however, the use cases and processes in the SUT are similar to real-world study information systems. The more important factor here is the selection of artificial defects that are representative of real-world projects. In the presented testbed, this selection is enabled by the possible introduction of more complex defects via the described mechanism of the defect clones.

Also, the size of the TbUIS system might limit its potential applicability as a benchmark for larger software systems. We are going to mitigate this concern by further evolution and extensions of the TbUIS.

V. Conclusion

In evaluating the effectiveness of testing techniques based on the measurement of the defect number that the test cases produced by these techniques detect in an experimental system, the established mutation testing approach can be accompanied by an alternative allowing the insertion of more complex defects caused by a misunderstanding of the design specification or other causes. We describe such an alternative in this paper: the presented TbUIS testbed, which is available as an open-source application and comprises a fictional university information system. The TbUIS testbed gives its user a mechanism to introduce artificial defects, including those from a predefined catalogue of possible defects, an extensive set of unit and FE-based functional automated tests, which can be used to examine test cases in the system, and a logging mechanism, which allows the collection of the data regarding which defects were activated by the examined test cases. Together with a good level of code and system documentation, the open structure of the TbUIS testbed eases its employment as a benchmark system to be used in the evaluation of path-based and combinatorial/constrained interaction testing techniques.

Acknowledgment

This work was supported by the European structural and investment funds (ESIF) project CZ.02.1.01/0.0/0.0/17_048/0007267 (InteCom)—Intelligent Components of Advanced Technologies for the Pilsen metropolitan area. Work package WP1.3: Methods and processes for control software safety assurance. The authors acknowledge the support of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 “Research Center for Informatics”.

References

[1] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness,” in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 351–360.

[2] J. Offutt, “A mutation carol: Past, present and future,” *Information and Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[3] C. Nie and H. Leung, “A survey of combinatorial testing,” *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[4] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, “Constrained interaction testing: A systematic literature study,” *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.

[5] M. Bures, “Pctgen: Automated generation of test cases for application workflows,” in *New Contributions in Information Systems and Technologies*. Cham: Springer International Publishing, 2015, pp. 789–794.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Program mutation: A new approach to program testing,” *Infotech State of the Art Report, Software Testing*, vol. 2, no. 1979, pp. 107–126, 1979.

[7] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: an automated class mutation system,” *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[8] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, “Assessment of class mutation operators for c++ with the mucpp mutation system,” *Information and Software Technology*, vol. 81, pp. 169–184, 2017.

[9] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, “Mutation testing advances: an analysis and survey,” in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[10] R. Gopinath, C. Jensen, and A. Groce, “Mutations: How close are they to real faults?” in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 189–200.

[11] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[12] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, “Experimental analysis of binary-level software fault injection in complex software,” in *2012 Ninth European Dependable Computing Conference*. IEEE, 2012, pp. 162–172.

[13] M. Kooli and G. Di Natale, “A survey on simulation-based fault injection tools for complex systems,” in *2014 9th IEEE International Conference On Design & Technology of Integrated Systems In Nanoscale Era (DTIS)*. IEEE, 2014, pp. 1–6.

[14] M. Bures, K. Frajtak, and B. S. Ahmed, “Tapir: Automation support of exploratory testing using model reconstruction of the system under test,” *IEEE Transactions on Reliability*, vol. 67, no. 2, pp. 557–580, 2018.

[15] M. Bures, “Model for evaluation and cost estimations of the automated testing architecture,” in *New Contributions in Information Systems and Technologies*. Cham: Springer International Publishing, 2015, pp. 781–787.

[16] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, “Threats to the validity of mutation-based test assessment,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 354–365.